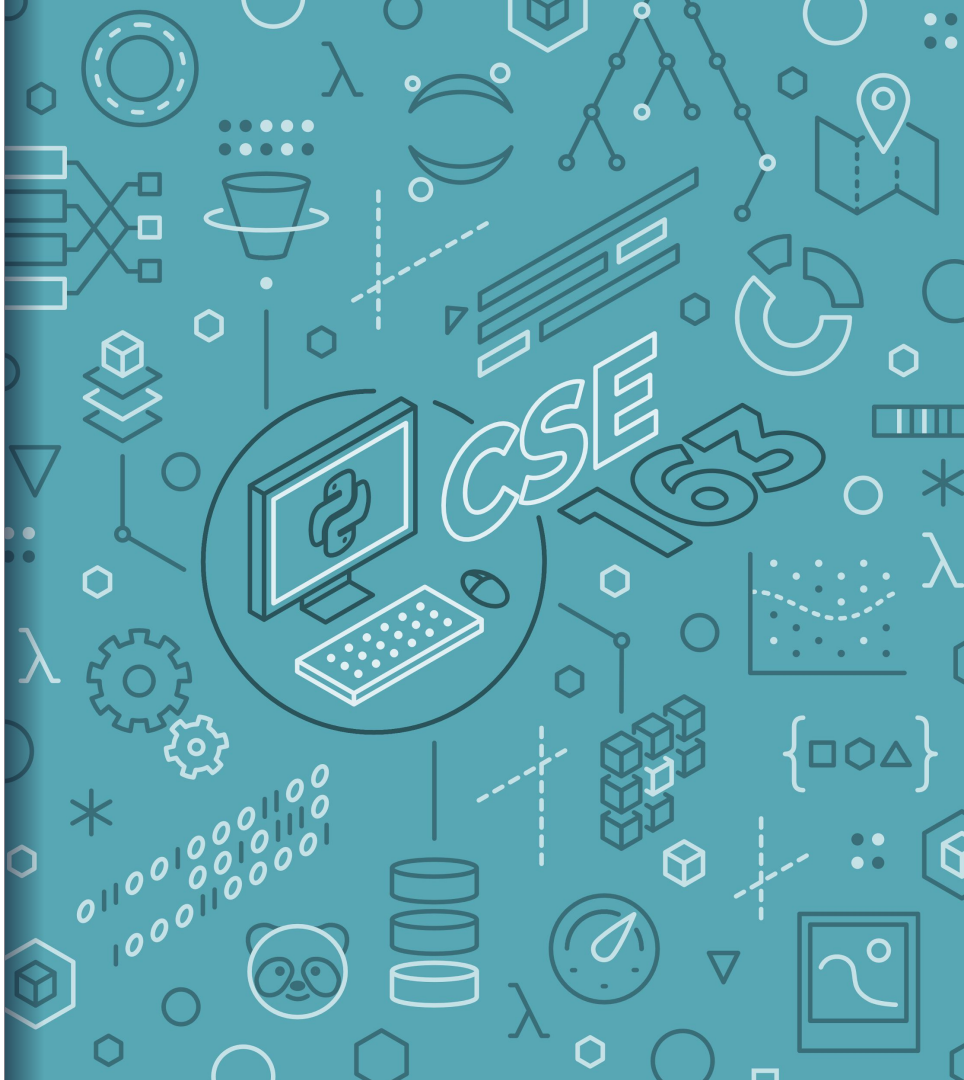
CSE 163

Hashing

Hunter Schafer



Week Agenda

- Monday
  - Hashing
- Wednesday
  - Exam Review
- Thursday
  - Exam Review
- Friday
  - **Exam in class**

Data Structures

- List: Series of values, each with an index
  - `list.append(v)` **O(1)**
  - `v in list` **O(n)**
  - `list[i]` **O(1)**
- Set: Collection of unique values, no sense of “order”
  - `set.add(v)` **O(1)**
  - `v in set` **O(1)**
  - `set[i]` **Not allowed**
- Dictionary: Like a set, but maps distinct keys to values
  - `dict[k] = v` **O(1)**
  - `k in dict` **O(1)**
  - `dict[k]` **O(1)**

How can the set/dictionary have run-times that don’t depend on the number of elements in the data structure?

Hashing

- The secret: Accessing an index in a list is fast
- Sets/Dictionaries are just lists behind the scenes
- Key ideas
  - Store a large list that is roomy (**“hash table”**)
  - Use a **“hash function”** to figure out the index for data
- A **hash function** is a function that takes any arbitrary data and turns it into a fixed-sized value (i.e. a number)
- We will focus on numbers as the data first

Set of Ints

- Hash Function: h(v) = v % 10

```
nums = set()
nums.add(11)  # 11 % 10 == 1
nums.add(49)  # 49 % 10 == 9
nums.add(24)  # 24 % 10 == 4
nums.add(7)   #  7 % 10 == 7
print(49 in nums)  # True
print(5 in nums)   # False
```

0	11	0	0	24	0	0	7	0	49
0	1	2	3	4	5	6	7	8	9

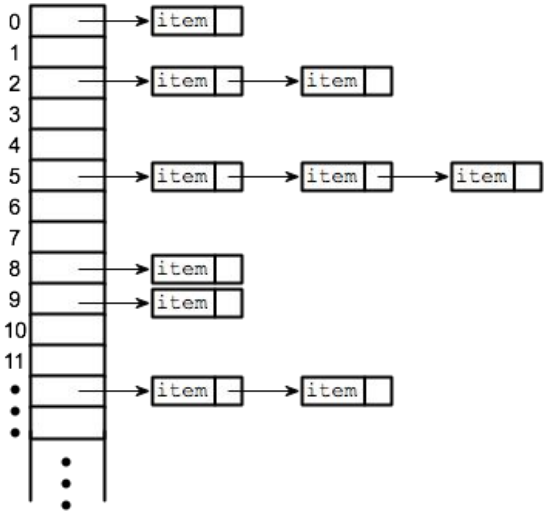
What value could I add that breaks this?

Collision

- This is nice and simple unless we get a collision
- A **collision** is when two values hash to the same index
  - Impossible to avoid when our hash table has a fixed size
- Two ways to make this work well in practice
  - Now: Have a way of dealing with collisions
  - Later: Come up with hash functions that avoid collisions as much as possible

Chained Hashing

- If we experience a collision, just store both values at that index
- Under some reasonable assumptions, these buckets will be small compared to the size of the whole set of items
- Care about the **load-factor** (num items / size of table)
  - If the load-factor is too high, these buckets will be big
  - Can rehash into a larger table to lower load-factor



Poll Everywhere

Think

1 min

pollev.com/cse163

If we do the insertions in the following order, what is the resulting hash table? Assume the following:

- The hash table has 4 slots
- If there is a collision, add the new value at front of the chain
- We are using the hash function: h(v) = v % 4

```
nums = set()
nums.add(11)
nums.add(43)
nums.add(21)
```

Poll Everywhere

Pair

2 min

pollev.com/cse163

If we do the insertions in the following order, what is the resulting hash table? Assume the following:

- The hash table has 4 slots
- If there is a collision, add the new value at front of the chain
- We are using the hash function: h(v) = v % 4

```
nums = set()
nums.add(11)
nums.add(43)
nums.add(21)
```

Hash Functions

- Our hash table can handle collisions, but we still want to keep the number of collisions low (otherwise buckets are large)
- Integers are relatively easy to hash, but what about strings?
- Each object can implement a __hash__(self) method
  - Write code to turn your object into an int

- What makes a good hash function?
  - Needs to return a number
  - Should minimize the number of collisions
  - Should “look random” to maximize spread of values

Hashing Strings

Attempt #1: Use the length of the string

```
hash('')     # 0
hash('hi')   # 2
hash('abc')  # 3
```

- Is this a good hash function?
  - Easy to make collisions
  - Length of string is concentrated for most strings

Hashing Strings

- Every character is represented by a number in the computer
  - ASCII: ' '=32, ..., 'a'=97, 'b'=98, …
- **Attempt #2:** Add up the ASCII values for each character

```
hash('')     # 0
hash('hi')   # 104 (h) + 105 (i) = 209
hash('dog')  # 97 (d) + 111 (o) + 103 (g) = 314
```

- Is this a good hash function?
  - Definitely better than **Attempt #1**!
  - Is it easy to make a collision?

Hashing Strings

- Last approach was good, but ignored crucial info: position
- Instead of just summing the values, add information about position too
- Many programming language (besides Python) use something like

$$hash(s) = \sum_{i=0}^{n-1} s[i] \cdot 31^{n-1-i}$$

- Python's is more complicated to avoid many real world attacks
- Basic idea
  - Don't throw information away

Hashing in Practice

- Hashing and equality are deeply related
  - If two values are equal, they must hash to same value
- If you implement __eq__ and __hash__, they need to be consistent
- Generally, people let Python make a good hash function

```
class Example:
    def __init__(self, a, b, c):
        # Initialize _a, _b, _c
    def __eq__(self, other):
        return self._a == other._a \
            and self._b = other._b
    def __hash__(self):
        return hash((self._a, self._b))
```

Brain Break

Hashing and Big Data

- Hashing is one of the key techniques used to help us work with large datasets.
- There are many different types of algorithms that scale that involve hashing (and many more that don't!)
- Today we are just going to focus on one that helps us count

Counting

- At our start-up Schmoogle, we provide a search engine for the world to use.
- We care a lot about our customer's privacy, but we also care about money so we want to figure out what the most searched terms are in order to better monetize our website
- Very simple way to track queries as they come in:
  - Store a dictionary with keys that are queries and values that are counts
  - How many keys would there be? Number of unique queries.
- At Schmoogle this is fine, but does this approach scale to the scale of our main competitor, Google?
  - They get trillions of searches a year, 15% of which are brand new and never seen before
  - That's a lot of keys

Count by Hash

- Idea: Instead of storing all the queries, use a hash table that stores counts.
- When a new query comes in (“dog”)
  - Hash it to find the index (4)
  - Increment the count at that index (77 -> 78)
- Now to find the count for a query (“cat”)
  - Hash it to find the index (6)
  - Return that number (17)
- Is the count of “cat” really 17?

14	302	94	124	77	400	17	0	19	43
0	1	2	3	4	5	6	7	8	9

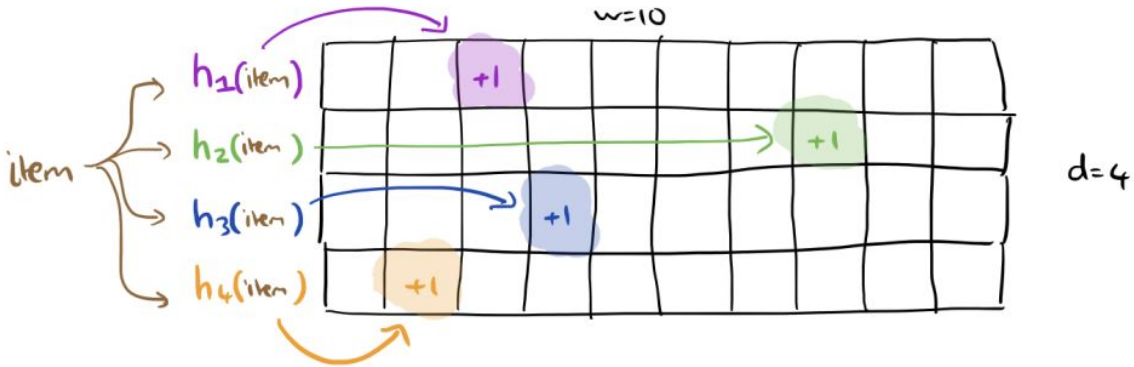
Sketching

- This technique is called a “sketch” of the data, it does not store all the data but lets us get the general idea
- Answers will be approximations of the true values
- Based on our counting scheme, the count we return is always an overestimate
  - It’s not possible to “miss” a count, but it is possible for a collision to overcount
- Collisions cause error in the counts, so to avoid them
  - Use a bigger table
  - Use a better hash function

- Very clever idea: Use multiple hash tables with multiple hash functions to reduce chance of collisions

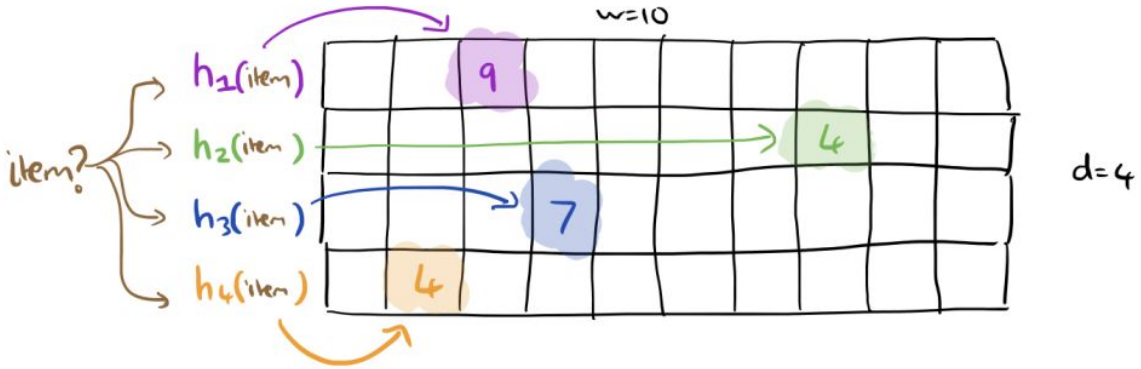
Count-Min Sketch

- Use **d** hash tables of size **w** with different hash functions
- Every time an item comes in, increment the count in each table
- Even if two values collide in table 1, it's unlikely they collide in table 2, 3, or 4!



Count-Min Sketch

- When we receive an item we look it up in all tables
- What result should we return?



Count-Min Sketch

- You can use this if the actual count is not of critical importance, but rather the rough idea is good enough
  - Good for page views, bad for counting payments
- You can't use this if the values can ever decrease
- Much better in terms of space!
  - A plain old hash-table needs O(n) space
  - The count-min-sketch needs O(wd) which can be much much smaller than n
- You can get provable guarantees on how large the error is or how big to make w and d, but the math can be pretty advanced
  - Challenge: You can read more here

Recap

- Important idea: Turn a value into something like a number and put it in a table
- We want to avoid collisions by using a good hash function, but they are inevitable so we must build in a way to handle them
- Applications of hashing
  - “Instant” lookup for sets/dictionaries
  - Approximate counts
  - Much much more!
    - Cryptography
    - Approximate nearest-neighbor
- I don’t want you to be able to implement a hash table from scratch, but I do want you to be able to understand the big idea